A METHOD OF MANAGING INFORMATION IN JAVA

The invention relates to a method of managing information in a distributed system context and using a remote invocation method of the Java language. The field of the invention is that of programming in a distributed system, and more specifically that of object-oriented programming in Java using the remote method invocation (RMI) mechanism.

BACKGROUND OF THE INVENTION

Object-oriented programming defines classes each having its own specific characteristics. Classes are interlinked by mother-daughter relations, a daughter class inheriting characteristics from its mother class. Characteristics include "variables" and "methods". A class is like a mold from which as many objects as necessary can be created.

Consider, for example, the vehicle class V for which variables (color, number of wheels, etc.) and methods (go forward, stop, etc.) are declared. Before an object from this class can be used, the programmer must instantiate it, i.e. create it by means of an instruction "new":

```
v = new V(green, 4, etc.)
```

By means of the above instruction the programmer creates an object v, which is also referred to as an instance, and which is a green vehicle with four wheels, etc. The methods of the class of the object can then be invoked, i.e. called in order to execute them.

If the programmer is working on a non-distributed system, he can instantiate an object directly, as described previously, and invoke the methods of that object.

One concept of the Java object-oriented programming language is referred to as an "interface". The inheritance rules for interfaces are different from those for classes. An interface does not define any non-constant variable and cannot be instantiated in the manner previously described. Initially the interface and

its methods are declared, but the methods are defined in a daughter class of the interface. Figure 1 shows the hierarchy of this inheritance in the case of two sister interfaces A and B from which a class C (ABlmpl) inherits for implementing the interfaces A and B. The arrows linking the classes and interfaces represent the inheritance relation.

The interfaces A and B themselves inherit from other interfaces D and E that will not be described in detail.

Thus, if the "display" method is declared for interface A and defined in class C (ABlmpl), for example, it can be invoked for the object a as follows:

```
A a = new ABlmpl ();
a.display();
```

To invoke the "print" method of interface B, the programmer can go via interface A using an instruction "horizontal casting" which entails considering a as an instance of class B, which is a sister of class A:

```
B b = (B)a;
b.print();
```

The semantic definition of the instruction "horizontal casting" is as follows (see Figure 2):

given three classes X, Y, Z and an instance z of class Z,

Z inherits directly or indirectly from X and from Y,

X does not inherit directly or indirectly from Y,

Y does not inherit directly or indirectly from X.

Horizontal casting is the operation of "casting" X on z in Y.

If the system is distributed, the classes and the interfaces are defined in a central server (remote system) and the instances are stored in a naming system by a naming service establishing a correspondence between a logical name and a remote object reference. In this example, after it has been instantiated in the server, the object v is stored in the naming system by means of the instruction:

registry (v, "greenvehicle");

The naming system is usually in a machine independent of the server. A distributed system includes one or more servers, a machine for the naming system and one or more local systems on a client site, which is generally remote from the site of the server. The server(s) and the local system(s) include one or more interfaces through which they communicate.

A programmer working on a client site cannot instantiate at this level an object of a class defined in the server, i.e. cannot use the instruction "new". The programmer must recover the object instantiated in the server and stored in the naming system and recovers it from the naming system by means of its reference. In this example, the programmer recovers the object v by means of its reference "greenvehicle" using the instruction:

V v = NamingService.get("greenvehicle");

The programmer can then invoke methods characteristic of the class of the object. These steps constitute the principal steps of using the remote method invocation (RMI) system, by means of which a client can recover the reference to a remote object from a local system in order to use the remote object. The RMI system includes a RMI interface and an implementation class of the RMI interface.

Figure 3 is a diagram showing a distributed system including a server S and a local system Cl. The RMI interfaces A and B and the class C (ABlmpl) for instantiating the objects of the interfaces A and B, as previously described, are defined in the server S. The interfaces A and B themselves inherit from the RMI interface, and objects instantiated by the server S can therefore be recovered by the system Cl. The client application "Client App" consists of a set of instructions; the arrows with black arrowheads between "Client App" and the interfaces A and B indicate that the

objects A and B used in "Client App" are recovered from the server S in which they are defined. The arrows with white arrowheads indicate inheritance relations and the arrows with black arrowheads indicate use relations.

However, in the case of interfaces of a distributed system, the instruction "horizontal casting" which then applies to a remote object, i.e. to an object whose methods are invoked via the RMI interface, cannot be used.

If the client site has more than one interface for using remote objects, the remote object reference recovered via one of them cannot be considered as a valid object reference for the other interfaces: to be able to use the same object from another interface, it is necessary to repeat the RMI steps previously described, which overloads the server and the naming service because it requires an additional record for each interface supported.

OBJECT AND SUMMARY OF THE INVENTION

The object of the invention is to facilitate the work of the programmer by rendering the distribution of the system transparent. The programmer can then recover a reference to a remote object in the same way as he would for a local object, especially in the case of interfaces.

The invention provides a method of managing information in a distributed system including at least one local system and one remote system and using a remote invocation method of the Java language, said language including instructions and enabling creation of objects from classes of belonging having hierarchical relations between them, said method consisting of defining in the local system classes replicating the hierarchy of classes in the remote system and including means of access to said classes in the remote system in order to enable use in the local system of instructions specific to classes defined in the remote system.

According to one feature of the invention one of the instructions is the instruction "horizontal casting".

The invention also provides a distributed information management system including at least one local system Cl and one remote system S including a plurality of interfaces A, B and using a remote method invocation mechanism of the Java language, said language including instructions and enabling creation of objects from classes of belonging, wherein the local system Cl includes a "proxy" PA, PB for each interface A, B and said proxy PA or PB is defined to enable use in the local system Cl of instructions specific to the interfaces A, B defined in the remote system S.

BRIEF DESCRIPTION OF THE DRAWINGS

Other features and advantages of the invention will become clearly apparent on reading the following description, which is given by way of non-limiting example and with reference to the accompanying drawings, in which:

Figure 1 shows an inheritance hierarchy between two interfaces and one class of a non-distributed system,

Figure 2 illustrates the semantic definition of the instruction "horizontal casting",

Figure 3 is a diagram showing the relations within a prior art distributed system including a server S and a local system Cl, and

Figure 4 shows the same system in accordance with the invention.

MORE DETAILED DESCRIPTION

As shown in Figure 4, the invention places between the client application "Client App" and the server S classes replicating in the system Cl the same inheritance hierarchy that applies between them in the server S. The gateway PA or "SmartProxyA" class corresponds to the interface A of the server S and the gateway PB or "SmartProxyB" class corresponds to the interface B. The classes PA and PB themselves inherit from other classes.

The classes PA and PB replicated in the system C1 are local and independent of the corresponding remote interfaces. This makes it possible to exploit the flexibility of programming based on local classes.

The "SmartProxy A" and "SmartProxy B" classes PA and PB effect by delegation the processing of the interfaces A and B, respectively. They encapsulate the references to the remote objects in the naming system.

In the context of a distributed system, the programmer therefore designs applications in a software context including an application programming interface (API) and a string of tools. In other words the applications are developed in accordance with specific rules as a function of the steps.

At the time of an instantiation, the objects are initialized with default values; a builder mechanism enables the programmer to define other initialization values.

A builder for recovering the references of the remote objects via the naming system is therefore defined in the "SmartProxy A" and "SmartProxy B" classes PA and PB. The instruction "new" can then be used to instantiate objects of the classes PA and PB defined locally in the system C1 but representing the interfaces A and B by delegation.

The instructions for locally invoking the "display" method of the interface A defined in the server in "Client App", namely:

```
A a = NamingService.get("referenceA");
a.display ();
```

become, in accordance with the invention:

```
PA sPA = new SmartProxyA;
SPA.display ();
```

The programmer can then use horizontal casting between the classes PA and PB defined locally to invoke a method of the interface B defined in the server, for example the "print" method:

```
PB sPB = (PB)sPA;
SPB.print ();
```